LOGIC and PROLOG

Richard Spencer-Smith

LOGIC AND PROLOG

Richard Spencer-Smith

New York London Toronto Sydney Tokyo Singapore

First published 1991 by
Harvester Wheatsheaf
66 Wood Lane End, Hemel Hempstead
Hertfordshire HP2 4RG
A division of
Simon & Schuster International Group

Printed in Great Britain at the
University Press, Cambridge

British Library Cataloguing in Publication Data

Spencer-Smith, Dr Richard
Logic and Prolog
1. Prolog Programming Language
1. Title
005.13

ISBN 0-7450-1022-9

1 2 3 4 5 95 94 93 92 91

Preface

In the production of this book, I owe a debt of gratitude in many quarters. First and foremost, to the University Grants Committee, without whose financial support it would not have been possible. The U.G.C, through its Computers in Teaching Initiative, provided funding for the development of a course combining Logic with Prolog.

In help with Prolog, I am particularly indebted to two people: Steve Torrance and Don Smith. I have been fortunate enough to have Steve as a close friend, and collaborator on other projects. Steve has shown me many things about Prolog, especially in relation to second order programming and production systems. He has been generous in supplying ideas for the book (e.g. the basis of Appendix 3), and has made many valuable comments on earlier drafts. I have also benefited enormously from the expert eye of Don Smith, who is responsible for the tail recursive definition of cumulative interest in section 5.2, and for a backwards chaining system which gave me the idea for the illustrative expert system of section 5.4. Amongst many who have given me help and encouragement, I would especially like to thank Ruth Crocket, Tony Drapkin, David Over and Alan Lacey for their comments.

I would also like to thank all those students at London, and latterly at Middlesex, who have suffered my various attempts to get this material right. On occasions they have shown me more elegant solutions than my own - both proofs and programs. Their

solutions than my own - both proofs and programs. Their criticisms have been the most valuable - of my mistakes, and incomprehensible formulations.

I have produced an Instructor's Manual to accompany the text. This arose initially out of my own need for a record of answers when marking - including acceptable alternative solutions to exercises, and common mistakes. Its development was further encouraged when I had postgraduate assistants to help with marking. Since many lecturers will have such assistance, I believe it is useful to make this material more widely available - but not so accessible as simply appearing in the back of the book. Any lecturer can obtain a free copy of this Manual from the publishers.

London, 1990.

Contents

Chapter 1
Introduction

1.1 Background to logic

What exactly is logic? Some will say that logic is centrally concerned with defining the notion of *proof* - an account of the inferential transformations that can lead to the derivation of a conclusion from a set of assumptions. Others may say that *logical truth*, and related notions involving meaning and truth, are central. Others may say that the definition of consistency is what logic is really all about. All of these concerns are important, but it is misleading to suggest that one is more fundamental than the others. One could equally well say that logic is essentially concerned with characterizing a few special expressions, known as the logical constants. It studies them from two angles: what they mean, and what inferential properties they have. This dual characterization of the logical constants only makes sense in the context of a general conception of meaning and of proof - that is, a context which includes those general relationships of truth, proof, consistency, etc.

Logic is relevant to any area of thinking or talking where the logical constants are used. Since these words are of the highest generality, that is really all areas. From theoretical physics to everyday talk of mundane matters, we make assertions and reason with sentences employing these words. This fact is sometimes expressed in the thesis that logic is ***topic-neutral***; it applies to all topics, and is partial towards none. But with some fields of inquiry logic enjoys a special bond. With linguistics there is a common interest in the structure of ordinary language, and the extent to which that is a logical structure. Psychologists are concerned with the nature of cognition; reasoning (however well or badly we ordinarily do it) is a cognitive process of central importance. To a

logical structure. Psychologists are concerned with the nature of cognition; reasoning (however well or badly we ordinarily do it) is a cognitive process of central importance. To a mathematician, symbolic logic is a branch of the subject like any other. Logical systems can be the object of mathematical investigation, and some of the most important theorems this century have concerned what can and cannot be done within them.

Computer science uses logic at its lowest level, in the switching of its circuitry, and at the highest level, in logic programming languages such as Prolog. However, with philosophy the link is at its most venerable. Philosophy gave birth to logic as a discipline, and still accords it a central place. Philosophy needs logic as a tool for the analysis of concepts and problems, and because its principal mode of demonstration is reasoned argument. Moreover, concepts such as truth, necessity and existence require both philosophical and formal investigation. What this list of disciplines indicates is that there are many different facets to the study and application of modern logic - so much so that it is impossible to begin to cover them all in a single work. This book is concerned with the *use* of logic, and especially its application in these last two subject areas.

Rather than launch straight in to the formalities, I will begin with some informal reasoning tasks. There is more to these exercises than appears at first sight, and we shall return to them in due course. You may have already encountered variants of the problems before. The kind I shall now describe are developed most fully in some books by Raymond Smullyan. In tribute to Smullyan's formulation, I shall call these ***knights*** and ***knaves*** problems. A knight is *honest*, in this very strict sense: everything he says is true. Knaves are correspondingly *dishonest*: anything a knave says is false. There is also the possibility of encountering a ***normal***: someone who, like normal people, sometimes speaks truly, sometimes falsely. There is a group of islands notorious for these characters. Some islands are populated only by knaves, others by knights, while some may even contain normals. People occasionally get washed up amongst these islands, and the locals are wont to have a little fun with them by venturing information about themselves - or disinformation, as the case may be. We shall assume in what follows that we are on the island of knights and knaves, i.e. that we will encounter both of these types, but no normals. (This is only an assumption, and may need to be revised later.)

A typical case is this. We come across two of the locals - let's call them 'a' and 'b'. a tells us that at least one of them is a knave. So the problem is: what, if anything, can we deduce about these

characters, from the information that a said this? Before proceeding to my account of the matter, you may like to stop and work out the solution for yourself.

Well, here's my version:

a says that at least one of a and b is a knave.
Suppose that a is a knave.
Then anything he says is false.
So it's false that at least one of them is a knave.
In other words, neither a nor b is a knave.
But then, in particular, a is not a knave.
This contradicts our initial hypothesis - so he can't be a knave.
If every islander is either knight or knave, a must be a knight.
If he's a knight, then what he said is true: at least one of them is a knave.
If at least one of them is a knave, and it isn't a, it must be b.
So we conclude: a is a knight and b a knave.

This sort of problem is in a sense rather artificial. In the present context this is an advantage, because it means that they are very self-contained. The assumptions which are relevant to the problem are few. A problem involving a more realistic example would tend to get us side-tracked into questions about what assumptions were correct and appropriate. With an artificial example we can concentrate on the reasoning, not the assumptions. It's better if there are a small number which can be agreed upon, for what's important is how one reasons *from* those assumptions. Many of the key words in the little specimen of reasoning above - 'suppose', 'if', 'not', 'every', 'and', 'so', 'true', 'or', 'false' - might occur in any piece of reasoning. They are the logical hinges on which an argument can turn, no matter what its topic. These are the sorts of words which logic is concerned to characterize, and thus our subject matter in what follows.

1.2 Background to Prolog

In order to do anything at all, a machine needs to be told what to do - either through some form of external control, as when operated by a person, or by containing within it a sequence of instructions which direct it - a ***program***. A machine might, for instance, compute the inference 'Socrates is human. Therefore: Socrates is mortal'. It can't do this without being programmed in some way to do so, e.g. by containing the instruction 'To prove that something mortal, show that it is

human'. For any computing machine it is customary to distinguish between its ***software***, the instructions and information which guide it, and its ***hardware***, the collection of devices which is the physical machine. The two are mutually independent. The same set of instructions may be physically embodied in totally different ways, in different machines. And the same machine may be able to follow many different kinds of instructions.

Instructions need to be formulated in a language. There are many computer languages, and different ones are suited to different tasks. They can be graded from ease of comprehension to humans (***high-level***), to ease of comprehension to computers (***low-level***). For every computer there is a fundamental language, its ***machine code***, the instructions of which correspond to the basic operations which it can perform (looking at the number stored in a certain memory location, say). At the other end of the scale are high-level languages like Prolog, which are relatively close to natural language. If a computer is running a Prolog program, in addition to the low-level description of what is going on, in machine code terms, there will also be a high-level description of what it is doing (drawing a certain inference, for example.).

A computer ***file*** can be a program, e.g. which tells the computer how to perform some function, such as word processing. Or a file could contain the data which such a program manipulates, e.g. the text of a document created by someone using the word processor. Some tasks which computers perform - airline bookings, say - require many particular items of data (flight details, passenger reservations, etc.) to be recorded in a very structured way, so that information can be accessed and manipulated in certain set ways - a ***database***. So a file might consist of a database of information. Or a file might contain the instructions which make it possible for the machine to be programmed in a particular language. Such a file would need to contain some means of translating a program written in that language, a ***source*** program, into the machine code of the computer, an ***object*** program. A file can be stored on whatever storage medium the machine uses - or by being written down on a piece of paper. Since a file is essentially a quantity of information, files are capable in principle of being transferred from one machine to another, e.g. over a telephone line.

One can distinguish two ways of approaching a programming task, two different programming styles: the ***imperative*** (or ***procedural***), and the ***descriptive*** (or ***declarative***). The imperative approach is epitomized by the question: how am I going to get the computer to do this task - what sequence of instructions should I give it?

This approach fits naturally with an imperative programming language - one in which many of the primitive expressions are *commands*. Most traditional computer languages have been imperative. The descriptive approach is characterized by a different attitude: how can I analyse and *describe* the matter in hand - the description to be in a form which the computer can use? The difference between the two styles can be illustrated by the kind of conditionals typically found in the corresponding programming languages. In an imperative language, they are like the conditional imperatives of ordinary language (e.g. 'If it's raining, put your coat on'). In a descriptive language, conditionals are like ordinary declarative conditionals (e.g. 'If it's raining, the pavements will be wet').

The logic programming language Prolog is typical of a descriptive language. However, even a descriptive language like Prolog has some imperative features. (In order to get the computer to display messages, for instance, we need something which is in effect an *instruction* to print.) This is why I prefer to draw the contrast primarily in terms of programming styles, rather than languages. Prolog counts as a descriptive language because it is suited to the descriptive attitude. Programming in it is an extension of our techniques for logical analysis and description. A problem is tackled by writing an analysis of it as a set of sentences in Prolog (a version of first order logic), which the machine can use like a set of axioms, to prove conclusions. A program is used not by telling it to run, but by putting questions to it - can a particular conclusion be derived? The overall approach, then, is one of description and deduction. But since we also have to pay attention to controlling the way deductions are carried out, and to producing side-effects like messages being printed out, there will usually be a procedural element too.

There is one other general contrast which is worth drawing - between two ways computers can be set up for programming. These ways are independent of the kind of language being used; rather, they are environments in which programming can take place. I shall say that the distinction is between ***prompt***-based systems, and ***windows***-based systems. The point of making this contrast is merely to indicate that there are different ways of entering, testing and changing a program, for it has a slight impact on the syntax of the language being used. A prompt-based system essentially allows just one point of contact with the language: the prompt. A multi-windows environment presents a more flexible interface to the user. Being easier to use, they generally require more sophisticated hardware on which to run. The distinction is neither exclusive (combinations of the two are possible)

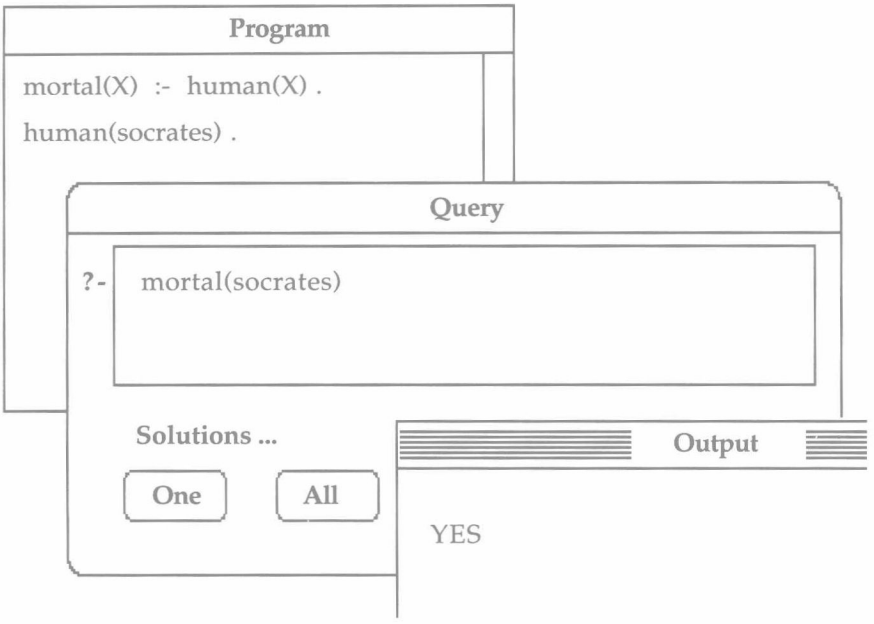
nor exhaustive (in the future, one might be able to give instructions to a computer by speaking to it in ordinary language).

The standard prompt in Edinburgh Prolog is the combination '?-'. When the prompt shows, one may interact with Prolog - telling it to add a certain sentence to its logical database, for instance, or asking it a question. Here is a short illustration of an interaction on a prompt based system:

```
?-    listing.
mortal(X) :- human(X) .
human(socrates).
yes
?-    mortal(socrates).
yes
?-
```

On the first of these lines, Prolog is told to show everything in its current program - to list all the sentences. There are just two: any X is mortal if it is human, and: Socrates is human. After these are displayed, the prompt reappears, and this time Prolog is queried: is it deducible that Socrates is mortal? Prolog makes the inference, gives the answer, and then returns the prompt, ready for the next interaction.

A multi-window environment is easier to use, because it can separate out some of these different aspects of programming into different windows. Here is roughly the same interaction in a windows system:



In the Program window at the top left, the program can be written - and edited like an ordinary piece of text on a word processor. This makes editing far simpler than having laboriously to enter edit instructions through the prompt. The middle window allows queries to be entered. The Output window, at the front, is where the solution to the query is displayed.

Lecturers at a university or college are likely to have access to a computer advisory service; those wishing to try out Prolog for the first time should consult the service as to whether their institution already has a licence for some version of Prolog; if not, whether it can be provided for the available machines; if not, which combination of hardware and software they would recommend. Those without access to a computer advisory could try looking in computer magazines for reviews of, and adverts for, implementations of Prolog; bear in mind that most commercial software (and hardware) is available to educational users at a substantial discount. It may be possible to obtain a demonstration version of the software for evaluation. In view of the above, one point to look out for is general ease of use - it's important that, for example, the business of editing doesn't get in the way of the more important task of working out what needs to be edited. Another

valuable feature in an educational context is the provision of detailed and intelligible error messages.

For those used only to teaching pure logic, the value of 'hands on' classes or workshops cannot be stressed enough; students need not only to have the general system demonstrated while they are sitting at a computer terminal, they also need practical sessions in Prolog programming and debugging. If the students are new to computers - or just new to the machines they will be using for Prolog - it is advisable to spend some time getting acquainted with the general environment before launching into Prolog. In particular, one should know how to edit, save, load and print out a file. A suitable exercise would be to use a word processing program with the same kind of interface as the Prolog implementation, to prepare some text - a letter, or an essay - and print it out.

1.3 Logic and Prolog

Many students of both logic and Prolog have great difficulty mastering the abstract concepts and techniques of these two subjects. This book attempts to explain these abstract concepts in an intuitively comprehensible way. It is thus aimed especially at those who are not well-versed in the use of formal or symbolic languages (unlike for example mathematicians and computer scientists) - students of philosophy, or cognitive science, or anyone studying Prolog as a first computer language. Despite their theoretical affinity, logic and Prolog are often not studied together. Nevertheless, studying one will enhance the learning of the other. The book is intended to provide a comprehensive grounding in both subjects, taking students from introductory level through to advanced features of Prolog, including some basic applications to artificial intelligence (A.I.). The relationship between logic and Prolog is developed from a practical, rather than theoretical perspective.

One component of philosophy, ancient and modern, is the analysis of concepts. Frege, the discoverer of modern formal logic, called it a *Begriffsschrift*, a concept notation. The analysis of a concept might involve attempting to get straight the conditions necessary and sufficient for its correct application, or seeing where such an attempt fails. For a simple example, see the analysis of knowledge, exercise 2.4, D. An analysis of knowledge involves a theory of the kind of thing which knowledge is, and may issue in a proposed definition of that concept. Logic serves the analysis and statement of such definitions and theories, making their content precise, e.g. by laying

bare the different readings of an ambiguous formulation, or showing exactly where qualifications or amendments are needed. One could say that philosophers have been using logic as a knowledge representation language longer than anyone. Analysis is one of the most important applications of logic in modern philosophy, yet most courses in logic do not seriously address it. At best they contain exercises in formalizing English sentences into logical formulae. But real problems do not come already packaged into a neat set of sentences; they are specified informally, and the difficult task is to produce a precise analysis from that informal description.

I have tried to design a number of logic exercises which give the student practice and confidence in applying logic to the analysis of problems. But problem solving is one area where logic *programming* has a great deal to offer the teaching of logic. In Prolog, you can work right through from an informal specification of a problem to a working analysis of it. You are not presented with a number of statements in English to translate into logical notation - getting a propositional description is half (or maybe much more) of the problem. You have to decide what the basic facts are, and what vocabulary you should use to describe them with, and then work out how to define other relationships in terms of that vocabulary - define the logical connections between these concepts. The computer makes the process of logical analysis interactive, providing very direct feedback in a way one simply cannot get if the sentences are just written down on paper. A student can see his or her logical definitions at work, and can get an idea from the answers of whether for example a defining condition needs to be added to or altered. Does the program prove fewer consequences than were intended - or more? Perhaps it works partially, answering some queries correctly but not others. One can experiment with it, and see the consequences of altering it in different ways.

Other examples of the pedagogical symbiosis between logic and Prolog are worth citing. Prolog can help to bring to life some aspects of the language of logic, for instance by showing that a variable is, precisely, variable: a term which can take various objects as its values. Recursion is another example. In elementary logic students learn techniques which are recursive, such as truth tables and proofs, without being explicitly introduced to the concept of recursion. By learning recursive programming in Prolog, they can come to a deeper understanding of these elementary recursions. Search, which is one of the major topics in A.I., provides another important example. By seeing proof construction as a search problem we can bring to bear some of the understanding of search developed in A.I., to provide students

with heuristics and strategies for reasoning. As a final example of the way Prolog can help with logic, I would mention the most important properties which can attributed to a logical system: that it be consistent (that it does not entail anything unwanted), and complete (that it does entail everything it should). I have not attempted to state proofs of any of the fundamental results involving these concepts, e.g. the consistency and completeness of propositional logic - material which, I believe, is neither necessary nor comprehensible to the average beginner. Rather, the approach here is practical: a student can get a much firmer grasp of what these properties really amount to by setting up and interpreting a small logical system of their own. The expert system sketched in sections 5.3 and 5.4 is just that: a simple logical system interpreted within Prolog.

Benefits also accrue in the other direction. At the elementary level, by providing students with a grounding in logic before attempting to program with logic, we can avoid some of the problems of adjusting to the declarative style of thinking typically encountered by those who come straight to Prolog from the traditional, procedural style of programming. Secondly, studying how proofs work (especially inferences involving Modus Ponens, Reductio, Disjunctive Syllogism and Universal Instantiation), helps to make clear what Prolog is actually doing when it solves a query - namely, performing a mechanical derivation according to such rules. As a final example: use of the more explicit notation of logic can help to clarify some of the more advanced features of Prolog - the precise force of such constructs as **not**, **forall** and **setof**.

Given all these potential benefits of studying the two subjects together, the question naturally arises as to how best they can be combined. Already they have grown apart in notation and terminology. This poses a dilemma: adopt a consistent style throughout, or mix the two? I have taken the latter course, since to do otherwise would be unfair to one or the other discipline, and it is my intention that this course should provide students with a firm grounding in both. There is a need for each community of practitioners to stay in touch with the other: for logicians to know about this most important area of application, and for logic programmers to know about the theoretical roots of their practice. Since it would be confusing to switch back and forth between the two disciplines too often, I have grouped them into two chapters apiece.

Texts and courses in logic tend to fall into two sorts. Those in informal logic look at real examples of reasoning, both valid and fallacious, but lack the rigour and generality of formal logic. Those in